Programming Languages

2nd edition

Tucker and Noonan

Chapter 3
Lexical and Syntactic Analysis

Syntactic sugar causes cancer of the semicolon.
A. Perlis

Contents

Lexical Analysis

Purpose: transform program representation

Input: printable Ascii characters

Output: tokens

Discard: whitespace, comments

Defn: A token is a logically cohesive sequence of characters representing a single symbol.

Example Tokens

Identifiers

Literals: 123, 5.67, 'x', true

Keywords: bool char ...

Operators: + - * / ...

Punctuation: ; , ( ) { }

Other Sequences

Whitespace: space tab

Comments

// any-char end-of-line*

End-of-line

End-of-file

Why a Separate Phase?

Simpler, faster machine model than parser

75% of time spent in lexer for non-optimizing compiler

Differences in character sets

End of line convention differs

Regular Expressions

RegExpr	Meaning
x	a character x
\x	an escaped character, e.g., \n
{ name }	a reference to a name
M | N	M or N
M N	M followed by N
M*	zero or more occurrences of M

RegExpr	Meaning
M+	One or more occurrences of M
M?	Zero or one occurrence of M
[aeiou]	the set of vowels
[0-9]	the set of digits
.	Any single character

Clite Lexical Syntax

Category	**Definition**
anyChar	[-~]
Letter	[a-zA-Z]
Digit	[0-9]
Whitespace	[\t]
Eol	\n
Eof	\004

Category	Definition
Keyword	bool | char | else | false | float | if | int | main | true | while
Identifier	{Letter}({Letter} | {Digit})*
integerLit	{Digit}+
floatLit	{Digit}+\.{Digit}+
charLit	‘{anyChar}’

Category	Definition
Operator	`=` | `||` | `&&` | `==` | `!=` | `<` | `<=` | `>` | `>=` | `+` | `-` | `*` | `/` | `!` | `[` | `]`
Separator	`:` | `.` | `{` | `}` | `(` | `)`
Comment	`//` ({anyChar} | {Whitespace})* {eol}

Generators

Input: usually regular expression

Output: table (slow), code

C/C++: Lex, Flex

Java: JLex

Finite State Automata

Set of states: representation – graph nodes

Input alphabet + unique end symbol

State transition function

Labelled (using alphabet) arcs in graph

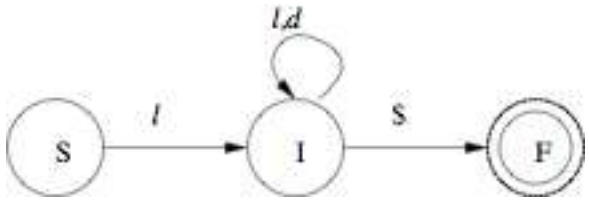
Unique start state

One or more final states

Deterministic FSA

Defn: A finite state automaton is *deterministic* if for each state and each input symbol, there is at most one outgoing arc from the state labeled with the input symbol.

A Finite State Automaton for Identifiers



Definitions

A *configuration* on an fsa consists of a state and the remaining input.

A *move* consists of traversing the arc exiting the state that corresponds to the leftmost input symbol, thereby consuming it. If no such arc, then:

- *If no input and state is final, then accept.*
- *Otherwise, error.*

An input is *accepted* if, starting with the start state, the automaton consumes all the input and halts in a final state.

Example

$$
\begin{aligned}
(S, a2i\$) &\vdash (I, 2i\$) \\
&\vdash (I, i\$) \\
&\vdash (I, \$) \\
&\vdash (F,)
\end{aligned}
$$

Thus: $(S, a2i\$) \vdash^* (F,)$

Some Conventions

Explicit terminator used only for program as a whole, not each token.

An unlabeled arc represents any other valid input symbol.

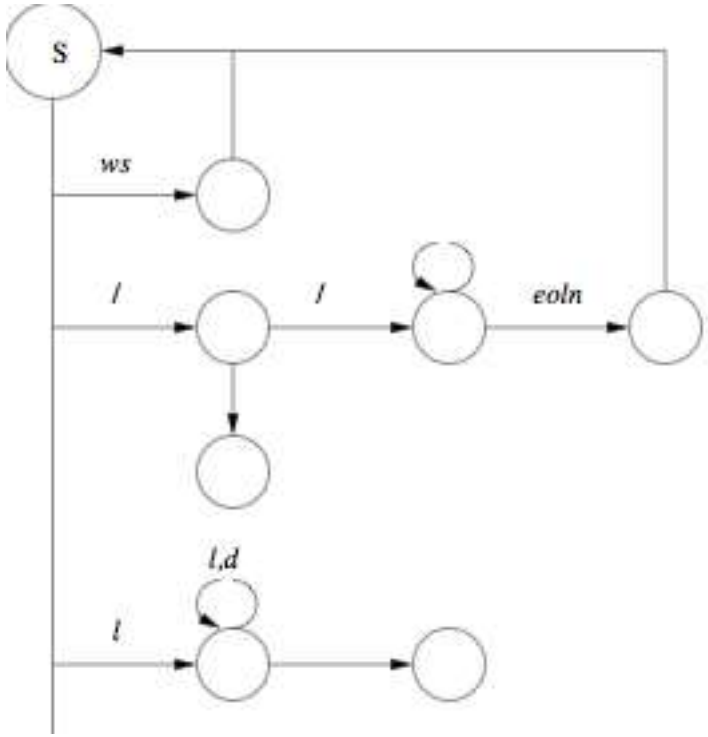
Recognition of a token ends in a final state.

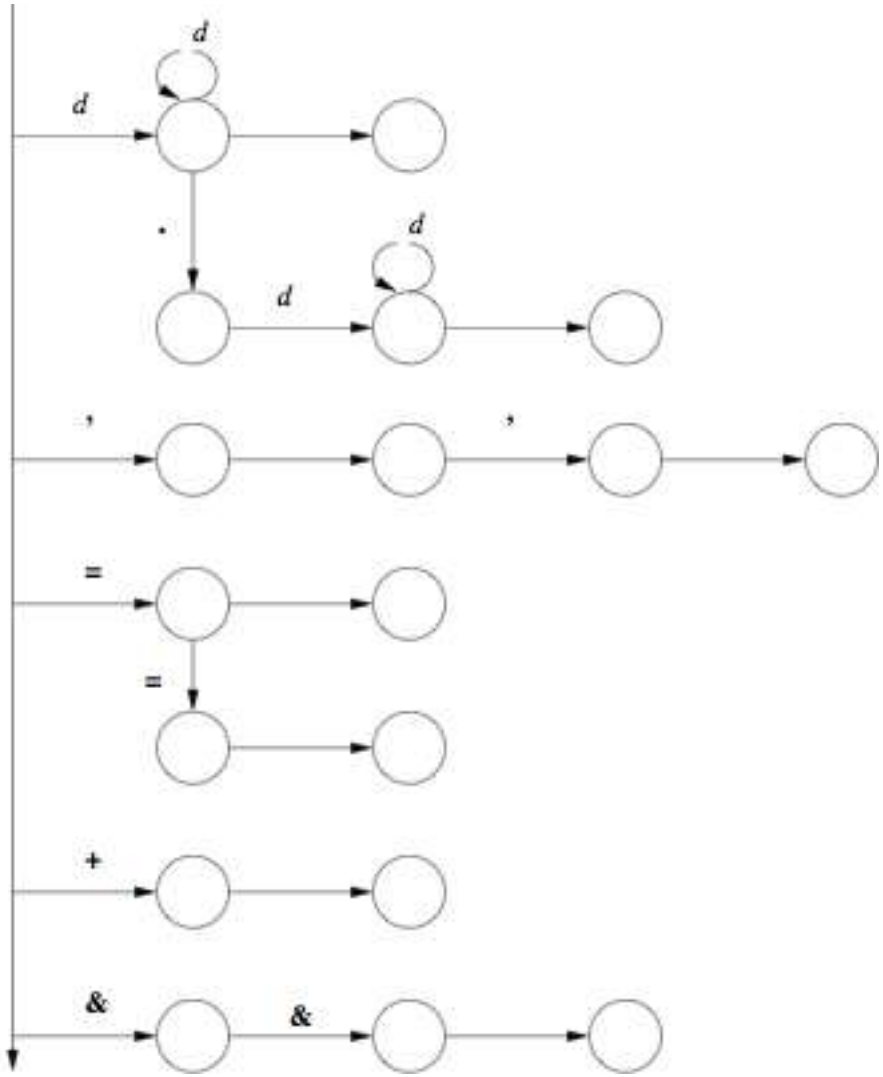
Recognition of a non-token transitions back to start state.

Recognition of end symbol (end of file) ends in a final state.

Automaton must be deterministic.

- *Drop keywords; handle separately.*
- *Must consider all sequences with a common prefix together.*


S
ws
l
l
eoln
l,d
l


d
d
.
d
d
,
,
=
=
+
&
&

Lexer Code

Parser calls lexer whenever it needs a new token.

Lexer must remember where it left off.

Greedy consumption goes 1 character too far

- *peek function*
- *pushback function*
- *no symbol consumed by start state*

From Design to Code

```
private char ch = ‘ ‘;
public Token next ( ) {
    do {
        switch (ch) {
            ...
        }
    } while (true);
}
```

Remarks

Loop only exited when a token is found

Loop exited via a return statement.

Variable ch must be global. Initialized to a space character.

Exact nature of a Token irrelevant to design.

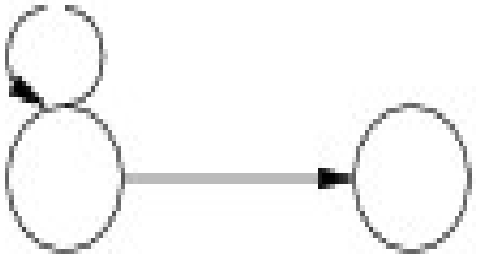
Translation Rules

Traversing an arc from A to B:

- *If labeled with x: test* ch == x
- *If unlabeled: else/default part of if/switch. If only arc, no test need be performed.*
- *Get next character if A is not start state*

A node with an arc to itself is a do-while.

- *Condition corresponds to whichever arc is labeled.*



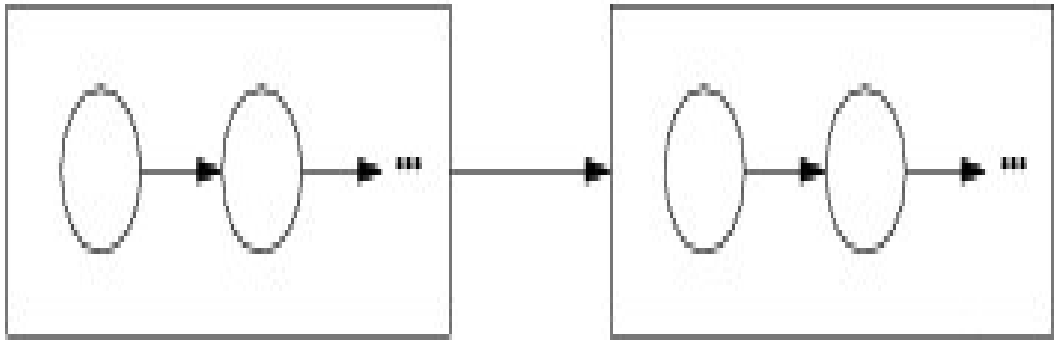
Otherwise the move is translated to a if/switch:

- *Each arc is a separate case.*
- *Unlabeled arc is default case.*

A sequence of transitions becomes a sequence of translated statements.

A complex diagram is translated by boxing its components so that each box is one node.

- *Translate each box using an outside-in strategy.*



```
private boolean isLetter(char c) {
    return ch >= ‘a’ && ch <= ‘z’ ||
        ch >= ‘A’ && ch <= ‘Z’;
}
```

```
private String concat(String set) {
    StringBuffer r = new StringBuffer("");
    do {
        r.append(ch);
        ch = nextChar( );
    } while (set.indexOf(ch) >= 0);
    return r.toString( );
}
```

```
public Token next( ) {
  do { if (isLetter(ch) { // ident or keyword
      String spelling = concat(letters+digits);
      return Token.keyword(spelling);
    } else if (isDigit(ch)) { // int or float literal
      String number = concat(digits);
      if (ch != '.')
        return Token.mkIntLiteral(number);
      number += concat(digits);
      return Token.mkFloatLiteral(number);
```

```
} else switch (ch) {
   case ‘ ‘: case ‘\t’: case ‘\r’: case eolnCh:
      ch = nextCh( );  break;
   case eofCh: return Token.eofTok;
   case ‘+’: ch = nextChar( );
      return Token.plusTok;
   …
   case ‘&’: check(‘&’); return Token.andTok;
   case ‘=‘: return chkOpt(‘=‘, Token.assignTok,
         Token.eqeqTok);
```

Source

```
// a first program
// with 2 comments
int main ( ) {
    char c;
    int i;
    c = 'h';
    i = c + 3;
} // main
```

Tokens

```
int
main
(
)
{
char
Identifier        c
;
```